Tutorial 4: 3D Model Rendering

Tutorial 4-1. Rendering 3D Model

This tutorial will cover how to render 3D models in DirectX 11 using HLSL. The code in this tutorial is based on the previous tutorial codes.

We have already been rendering 3D models in the previous tutorials; however, they were composed of a single triangle and were uninteresting. Now that the basics have been covered, we'll move forward to render a more complex object. In this case, the object will be a cube. Before we get into how to render more complex models, we will first talk about model formats.

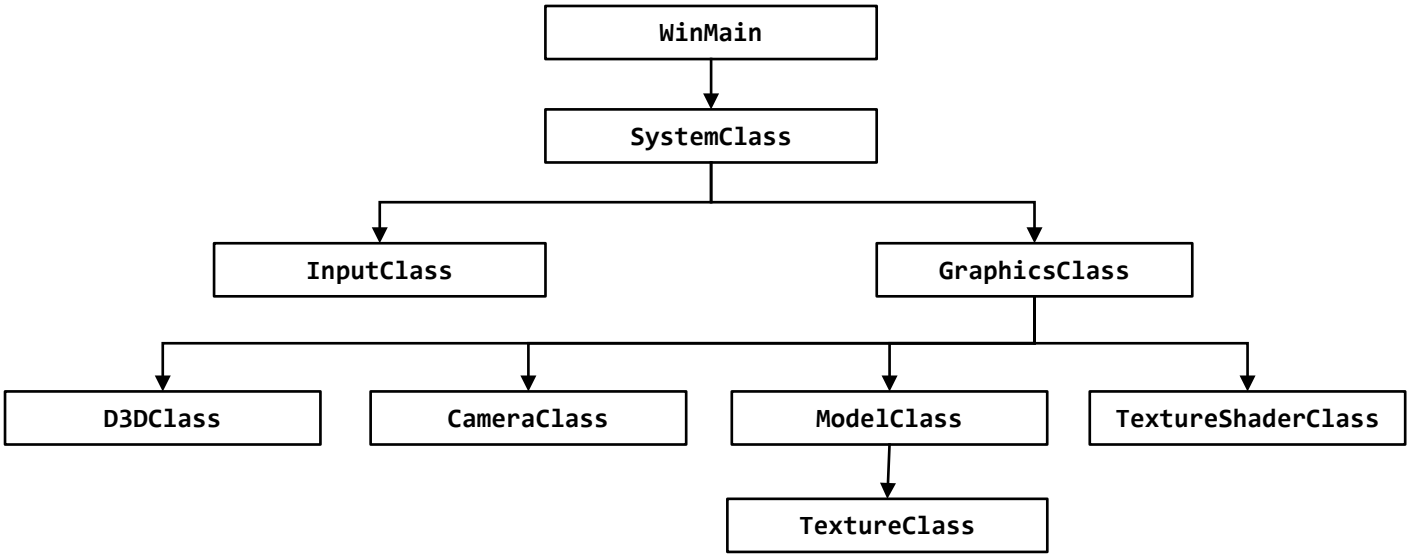
There are many tools available that allow users to create 3D models. Maya and 3D Studio Max are two of the more popular 3D modeling programs. There are also many other tools with less features but still can do the basics for what we need.

Regardless of which tool you choose to use they will all export their models into numerous different formats. My suggestion is that you create your own model format and write a parser to convert their export format into your own format. The reason for this is that the 3D modeling package that you use may change over time and their model format will also change. Also, you may be using more than one 3D modeling package so you will have multiple different formats to deal with. So, if you have your own format and convert their formats to your own then your code will never need to change. You will only need to change your parser program(s) for changing those formats to your own. As well most 3D modeling packages export a ton of junk that is only useful to that modeling program and you don't need any of it in your model format.

The most important part to making your own format is that it covers everything you need it to do and that it is simple for you to use. You can also consider making a couple different formats for different objects as some may have animation data, some may be static, and so forth.

The model format I'm going to present is very basic. It will contain a line for each vertex in the model. Each line will match the vertex format used in the code which will be position vector (x, y, z), texture coordinates (tu, tv), and the normal vector (nx, ny, nz). The format will also have the vertex count at the top so you can read the first line and build the memory structures needed before reading in the data. The format will also require that every three lines makes a triangle, and that the vertices in the model format are presented in clockwise order. Here is the model file for the cube we are going to render:

Framework



Cube.txt

Vertex Count: 36

Data:

```
-1.0  1.0 -1.0 0.0 0.0  0.0  0.0 -1.0
 1.0  1.0 -1.0 1.0 0.0  0.0  0.0 -1.0
-1.0 -1.0 -1.0 0.0 1.0  0.0  0.0 -1.0
-1.0 -1.0 -1.0 0.0 1.0  0.0  0.0 -1.0
 1.0  1.0 -1.0 1.0 0.0  0.0  0.0 -1.0
 1.0 -1.0 -1.0 1.0 1.0  0.0  0.0 -1.0
 1.0  1.0 -1.0 0.0 0.0  1.0  0.0  0.0
 1.0  1.0  1.0 1.0 0.0  1.0  0.0  0.0
 1.0 -1.0 -1.0 0.0 1.0  1.0  0.0  0.0
 1.0 -1.0 -1.0 0.0 1.0  1.0  0.0  0.0
 1.0  1.0  1.0 1.0 0.0  1.0  0.0  0.0
 1.0 -1.0  1.0 1.0 1.0  1.0  0.0  0.0
 1.0  1.0  1.0 0.0 0.0  0.0  0.0  1.0
-1.0  1.0  1.0 1.0 0.0  0.0  0.0  1.0
 1.0 -1.0  1.0 0.0 1.0  0.0  0.0  1.0
 1.0 -1.0  1.0 0.0 1.0  0.0  0.0  1.0
-1.0  1.0  1.0 1.0 0.0  0.0  0.0  1.0
-1.0 -1.0  1.0 1.0 1.0  0.0  0.0  1.0
-1.0  1.0  1.0 0.0 0.0 -1.0  0.0  0.0
-1.0  1.0 -1.0 1.0 0.0 -1.0  0.0  0.0
-1.0 -1.0  1.0 0.0 1.0 -1.0  0.0  0.0
-1.0 -1.0  1.0 0.0 1.0 -1.0  0.0  0.0
-1.0  1.0 -1.0 1.0 0.0 -1.0  0.0  0.0
-1.0 -1.0 -1.0 1.0 1.0 -1.0  0.0  0.0
-1.0  1.0  1.0 0.0 0.0  0.0  1.0  0.0
 1.0  1.0  1.0 1.0 0.0  0.0  1.0  0.0
-1.0  1.0 -1.0 0.0 1.0  0.0  1.0  0.0
-1.0  1.0 -1.0 0.0 1.0  0.0  1.0  0.0
 1.0  1.0  1.0 1.0 0.0  0.0  1.0  0.0
 1.0  1.0 -1.0 1.0 1.0  0.0  1.0  0.0
-1.0 -1.0 -1.0 0.0 0.0  0.0 -1.0  0.0
 1.0 -1.0 -1.0 1.0 0.0  0.0 -1.0  0.0
-1.0 -1.0  1.0 0.0 1.0  0.0 -1.0  0.0
-1.0 -1.0  1.0 0.0 1.0  0.0 -1.0  0.0
 1.0 -1.0 -1.0 1.0 0.0  0.0 -1.0  0.0
 1.0 -1.0  1.0 1.0 1.0  0.0 -1.0  0.0
```

So, as you can see there are 36 lines of x, y, z, tu, tv, nx, ny, nz data. Every three lines composes its own triangle giving us 12 triangles that will form a cube. The format is very straight forward and can be read directly into our vertex buffers and rendered without any modifications.

Now one thing to watch out for is that some 3D modeling programs export the data in different orders such as left-hand or right-hand coordinate systems. Remember that by default DirectX 11 is a left-handed coordinate system by default and so the model data needs to match that. Keep an eye out for those differences and ensure your parsing program can handle converting data into the correct format/order.

modelclass.h

For this tutorial all we needed to do was make some minor changes to the ModelClass for it to render 3D models from our text model files.

The fstream library is now included to handle reading from the model text file.

[Same as previous codes]

```
#include <fstream>
using namespace std;

class ModelClass
```

```
{
private:
```

The VertexType structure now has a **normal** vector (a line that is perpendicular to the face of the polygon) to accommodate lighting, which will be used in the lighting tutorial.

```
	struct VertexType
	{
		D3DXVECTOR3 position;
		D3DXVECTOR2 texture;
		D3DXVECTOR3 normal;
	};
```

The next change is the addition of a new structure to represent the model format. It is called ModelType. It contains position, texture, and normal vectors the same as our file format does.

```
	struct ModelType
	{
		float x, y, z;
		float tu, tv;
		float nx, ny, nz;
	};
```

The Initialize function will now take as input the character string file name of the model to be loaded.

```
public:
	~~bool Initialize(ID3D11Device*, WCHAR*);~~
	bool Initialize(ID3D11Device*, char*, WCHAR*);
```

We also have two new functions to handle loading and unloading the model data from the text file.

```
	bool LoadModel(char*);
	void ReleaseModel();
```

The final change is a new private variable called m_model which is going to be an array of the new private structure ModelType. This variable will be used to read in and hold the model data before it is placed in the vertex buffer.

```
private:
	ModelType* m_model;
};
```

modelclass.cpp

```
ModelClass::ModelClass()
{
```

The new model structure is set to null in the class constructor.

```
	m_model = 0;
}
```

The Initialize function now takes as input the file name of the model that should be loaded.

```
~~bool ModelClass::Initialize(ID3D11Device* device, WCHAR* textureFilename)~~
bool ModelClass::Initialize(ID3D11Device* device, char* modelFilename, WCHAR* textureFilename)
{
```

In the Initialize function we now call the new LoadModel function first. It will load the model data from the file name we provide into the new m_model array. Once this model array is filled, we can then build the vertex and index buffers from it. Since InitializeBuffers now depends on this model data you must make sure to call the functions in the correct order.

```
	bool result;
```

```
	// Load in the model data,
	result = LoadModel(modelFilename);
	if(!result)
	{
		return false;
	}
}

void ModelClass::Shutdown()
{
```

In the Shutdown function we add a call to the ReleaseModel function to delete the m_model array data once we are done.

```
	// Release the model data.
	ReleaseModel();

	return;
}

bool ModelClass::InitializeBuffers(ID3D11Device* device)
{
```

Take note that we will no longer manually set the vertex and index count here. Once we get to the ModelClass::LoadModel function you will see that we read the vertex and index counts in at that point instead.

```
	// Set the number of vertices in the vertex array.
	m_vertexCount = 3;

	// Set the number of indices in the index array.
	m_indexCount = 3;

	// Create the vertex array.
	vertices = new VertexType[m_vertexCount];
	if(!vertices)
	{
		return false;
	}

	// Create the index array.
	indices = new unsigned long[m_indexCount];
	if(!indices)
	{
		return false;
	}

	// Load the vertex array with data.
	vertices[0].position = D3DXVECTOR3(-1.0f, -1.0f, 0.0f);  // Bottom left.
	vertices[0].texture = D3DXVECTOR2(0.0f, 1.0f);

	vertices[1].position = D3DXVECTOR3(0.0f, 1.0f, 0.0f);  // Top middle.
	vertices[1].texture = D3DXVECTOR2(0.5f, 0.0f);

	vertices[2].position = D3DXVECTOR3(1.0f, -1.0f, 0.0f);  // Bottom right.
	vertices[2].texture = D3DXVECTOR2(1.0f, 1.0f);

	// Load the index array with data.
	indices[0] = 0;  // Bottom left.
	indices[1] = 1;  // Top middle.
	indices[2] = 2;  // Bottom right.
```

Loading the vertex and index arrays has changed a bit. Instead of setting the values manually we loop through all the elements in the new m_model array and copy that data from there into the vertex array. The index array is easy to build as each vertex we load has the same index number as the position in the array it was loaded into.

```
	int i;
```

```
	// Load the vertex array and index array with data.
	for(i=0; i<m_vertexCount; i++)
	{
		vertices[i].position = D3DXVECTOR3(m_model[i].x, m_model[i].y, m_model[i].z);
		vertices[i].texture = D3DXVECTOR2(m_model[i].tu, m_model[i].tv);
		vertices[i].normal = D3DXVECTOR3(m_model[i].nx, m_model[i].ny, m_model[i].nz);

		indices[i] = i;
	}
}
```

This is the new LoadModel function which handles loading the model data from the text file into the m_model array variable. It opens the text file and reads in the vertex count first. After reading the vertex count it creates the ModelType array and then reads each line into the array. Both the vertex count and index count are now set in this function.

```
bool ModelClass::LoadModel(char* filename)
{
	ifstream fin;
	char input;
	int i;


	// Open the model file.
	fin.open(filename);

	// If it could not open the file then exit.
	if(fin.fail())
	{
		return false;
	}

	// Read up to the value of vertex count.
	fin.get(input);
	while(input != ':')
	{
		fin.get(input);
	}

	// Read in the vertex count.
	fin >> m_vertexCount;

	// Set the number of indices to be the same as the vertex count.
	m_indexCount = m_vertexCount;

	// Create the model using the vertex count that was read in.
	m_model = new ModelType[m_vertexCount];
	if(!m_model)
	{
		return false;
	}

	// Read up to the beginning of the data.
	fin.get(input);
	while(input != ':')
	{
		fin.get(input);
	}
	fin.get(input);
	fin.get(input);

	// Read in the vertex data.
	for(i=0; i<m_vertexCount; i++)
	{
		fin >> m_model[i].x >> m_model[i].y >> m_model[i].z;
		fin >> m_model[i].tu >> m_model[i].tv;
```

```
            fin >> m_model[i].nx >> m_model[i].ny >> m_model[i].nz;
        }

        // Close the model file.
        fin.close();

        return true;
}
```

The ReleaseModel function handles deleting the model data array.

```
void ModelClass::ReleaseModel()
{
        if(m_model)
        {
            delete [] m_model;
            m_model = 0;
        }

        return;
}
```

graphicsclass.h

```
class GraphicsClass
{
```

Render now takes a float value as input.

```
private:
        bool Render();
        bool Render(float);
};
```

graphicsclass.cpp

```
bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

The model initialization now takes in the filename of the model file it is loading. In this tutorial, we will use the cube.txt file so this model loads in a 3D cube object for rendering.

```
        // Initialize the model object.
        result = m_Model->Initialize(m_D3D->GetDevice(), L"../Engine/data/seafloor.dds");
        result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/cube.txt", L"../Engine/data/seafloor.dds");
}

bool GraphicsClass::Frame()
{
```

We add a new static variable to hold an updated rotation value each frame that will be passed into the Render function.

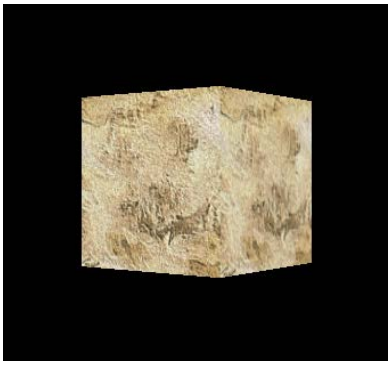
```
        bool result;

        static float rotation = 0.0f;

        // Update the rotation variable each frame.
        rotation += (float)D3DX_PI * 0.01f;
        if(rotation > 360.0f)
        {
            rotation -= 360.0f;
        }
```

```
        // Render the graphics scene.
        result = Render();
        result = Render(rotation);
}

bool GraphicsClass::Render()
bool GraphicsClass::Render(float rotation)
{
```

Here we rotate the world matrix by the rotation value so that when we render the cube using this updated world matrix it will spin the cube by the rotation amount.

```
        // Rotate the world matrix by the rotation value so that the triangle will spin.
        D3DXMatrixRotationY(&worldMatrix, rotation);

        // Put the model vertex and index buffers on the graphics pipeline to prepare them for drawing.
        m_Model->Render(m_D3D->GetDeviceContext());
}
```

Summary



With the changes to the ModelClass, we can now load in 3D models and render them. The format used here is just for basic static objects; however, it is a good start to understanding how model formats work.

Exercise

1. Add a different 3D models (e.g. pyramid, cylinder, sphere, etc…) to the current scene. Each 3D model should be textured differently from other models.
2. Rotate each model with different orientations.

Tutorial 4-2. Loading .OBJ Models

This tutorial will cover how to import static 3D models from a modeling tool (i.e. Maya or 3D Max). Note that this tutorial will be focused on .obj models but it also applies to pretty much any other 3D modeling software package with some slight changes.

In the previous tutorials, we have already created our own model format and rendered 3D models using that format. The goal now is to convert .obj models into our format and render them. we won't go into how to model 3D objects in a modeling tool as there are hundreds of tutorials on the net already dedicated to that, we will instead start at the point where you have a textured and triangulated 3D model ready for export.

For the 3D model format, we will use the .obj format as it is easily readable and good for beginners to start with.

To export your model in the .obj format you must first enable the .obj exporter in Maya or 3D Max. Click "Window", then "Settings/Preferences", then "Plug-in Manager". Scroll down to objExport.mll and select both "Loaded" and "Auto load". Now to export your model in this format click on "File", then "Export All". Now at the bottom select "Files of type:" and scroll down and select "OBJexport". Give it a file name and hit "Export All" and it will export it to a text file with a .obj extension. To look at the file you can right click and select Open With and choose WordPad to read the file. You will then see something that looks like the following:

Cube.obj

```
# This file uses centimeters as units for non-parametric coordinates.

mtllib cube.mtl
g default
v -0.500000 -0.500000 0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 0.500000 0.500000
v 0.500000 0.500000 0.500000
v -0.500000 0.500000 -0.500000
v 0.500000 0.500000 -0.500000
v -0.500000 -0.500000 -0.500000
v 0.500000 -0.500000 -0.500000
vt 0.001992 0.001992
vt 0.998008 0.001992
vt 0.001992 0.998008
vt 0.998008 0.998008
vt 0.001992 0.001992
vt 0.998008 0.001992
vt 0.001992 0.998008
vt 0.998008 0.998008
vt 0.001992 0.001992
vt 0.998008 0.001992
vt 0.001992 0.998008
vt 0.998008 0.998008
vt 0.001992 0.001992
vt 0.998008 0.001992
vt 0.001992 0.998008
vt 0.998008 0.998008
vt 0.001992 0.001992
vt 0.998008 0.001992
vt 0.001992 0.998008
vt 0.998008 0.998008
vt 0.998008 0.998008
vt 0.001992 0.998008
vt 0.998008 0.001992
vt 0.001992 0.001992
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 1.000000 0.000000
```

```
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
s 1
g pCube1
usemtl file1SG
f 1/1/1 2/2/2 3/3/3
f 3/3/3 2/2/2 4/4/4
s 2
f 3/13/5 4/14/6 5/15/7
f 5/15/7 4/14/6 6/16/8
s 3
f 5/21/9 6/22/10 7/23/11
f 7/23/11 6/22/10 8/24/12
s 4
f 7/17/13 8/18/14 1/19/15
f 1/19/15 8/18/14 2/20/16
s 5
f 2/5/17 8/6/18 4/7/19
f 4/7/19 8/6/18 6/8/20
s 6
f 7/9/21 1/10/22 5/11/23
f 5/11/23 1/10/22 3/12/24
```

This .obj model file represents a 3D cube. It has 8 vertices, 24 texture coordinates and normal vectors, and 6 sides made up of 12 faces in total. When examining the file you can ignore every line unless it starts with a "V", "VT", "VN", or "F". The extra information in the file will not be needed for converting .obj to our file format. Lets look at what each of the important lines means:

1. The "V" lines are for the vertices. The cube is made up of 8 vertices for the eight corners of the cube. Each is listed in X, Y, Z float format.

2. The "VT" lines are for the texture coordinates. The cube is has 24 texture coordinates and most of them are duplicated since it records them for every vertex in every triangle in the cube model. They are listed in TU, TV float format.

3. The "VN" lines are for the normal vectors. The cube is has 24 normal vectors and most of them are duplicated again since it records them for every vertex in every triangle in the cube model. They are listed in NX, NY, NZ float format.

4. The "F" lines are for each triangle (face) in the cube model. The values listed are indexes into the vertices, texture coordinates, and normal vectors. The format of each face is:

f Vertex1/Texture1/Normal1 Vertex2/Texture2/Normal2 Vertex3/Texture3/Normal3

So, a line that says "f 3/13/5 4/14/6 5/15/7" then translates to "Vertex3/Texture13/Normal5 Vertex4/Texture14/Normal6 Vertex5/Texture15/Normal7".

The order the data is listed in the .obj file is very important. For example, the first vertex in the file corresponds to Vertex1 in the face list. This is the same for texture coordinates and normals as well.

Looking at the face lines in the .obj file notice that the three index groups per line make an individual triangle. And in the case of this cube model the 12 total faces make up the 6 sides of the cube that has 2 triangles per side.

Right hand to Left hand conversion

By default, Maya or 3D max is a right-handed coordinate system and exports the .obj file data in right hand coordinates. To convert that data into a left-handed system which DirectX 11 is by default you have to do the following:

1. Invert the Z coordinate vertices. In the code you will see it do this: vertices[vertexIndex].z = vertices[vertexIndex].z * -1.0f;

2. Invert the TV texture coordinate. In the code you will see it do this: texcoords[texcoordIndex].y = 1.0f - texcoords[texcoordIndex].y;

3. Invert the NZ normal vertex. In the code you will see it do this: normals[normalIndex].z = normals[normalIndex].z * -1.0f;

4. Convert the drawing order from counter clockwise to clockwise. In the code we simply read in the indexes in reverse order instead of re-organizing it after the fact:

```
fin >> faces[faceIndex].vIndex3 >> input2 >> faces[faceIndex].tIndex3 >> input2 >> faces[faceIndex].nIndex3;
fin >> faces[faceIndex].vIndex2 >> input2 >> faces[faceIndex].tIndex2 >> input2 >> faces[faceIndex].nIndex2;
fin >> faces[faceIndex].vIndex1 >> input2 >> faces[faceIndex].tIndex1 >> input2 >> faces[faceIndex].nIndex1;
```

With those four steps complete the model data will be ready for DirectX 11 to render correctly.

main.cpp

The program to convert the .obj files into our DirectX 11 format is fairly simple and is a single program file called main.cpp. It opens a command prompt and asks for the name of the .obj file to convert. Once the user types in the name it will attempt to open the file and read in the data counts and build the structures required to read the data into. After that it reads the data into those structures and converts it to a left hand system. Once that is done it then writes the data out to a model.txt file. That file can then be renamed and used for rendering in DirectX 11 using the 3D model render project from the previous tutorial.

```
#include <iostream>
#include <fstream>
using namespace std;

// Type definitions
typedef struct
{
	float x, y, z;
}VertexType;

typedef struct
{
	int vIndex1, vIndex2, vIndex3;
	int tIndex1, tIndex2, tIndex3;
	int nIndex1, nIndex2, nIndex3;
}FaceType;

// Function prototypes
void GetModelFilename(char*);
bool ReadFileCounts(char*, int&, int&, int&, int&);
bool LoadDataStructures(char*, int, int, int, int);

int main()
{
	bool result;
	char filename[256];
	int vertexCount, textureCount, normalCount, faceCount;
	char garbage;
```

```
	// Read in the name of the model file.
	GetModelFilename(filename);

	// Read in the number of vertices, tex coords, normals, and faces so that the data structures can be initialized with the exact sizes needed.
	result = ReadFileCounts(filename, vertexCount, textureCount, normalCount, faceCount);
	if(!result)
	{
		return -1;
	}

	// Display the counts to the screen for information purposes.
	cout << endl;
	cout << "Vertices: " << vertexCount << endl;
	cout << "UVs:      " << textureCount << endl;
	cout << "Normals:  " << normalCount << endl;
	cout << "Faces:    " << faceCount << endl;

	// Now read the data from the file into the data structures and then output it in our model format.
	result = LoadDataStructures(filename, vertexCount, textureCount, normalCount, faceCount);
	if(!result)
	{
		return -1;
	}

	// Notify the user the model has been converted.
	cout << "\nFile has been converted." << endl;
	cout << "\nDo you wish to exit (y/n)? ";
	cin >> garbage;

	return 0;
}

void GetModelFilename(char* filename)
{
	bool done;
	ifstream fin;

	// Loop until we have a file name.
	done = false;
	while(!done)
	{
		// Ask the user for the filename.
		cout << "Enter model filename: ";

		// Read in the filename.
		cin >> filename;

		// Attempt to open the file.
		fin.open(filename);

		if(fin.good())
		{
			// If the file exists and there are no problems then exit since we have the file name.
			done = true;
		}
		else
		{
			// If the file does not exist or there was an issue opening it then notify the user and repeat the process.
			fin.clear();
			cout << endl;
			cout << "File " << filename << " could not be opened." << endl << endl;
		}
	}

	return;
}
```

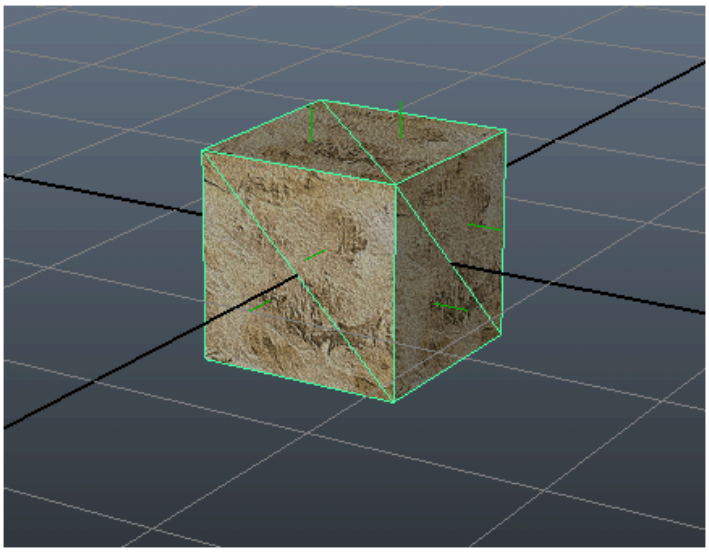
```
bool ReadFileCounts(char* filename, int& vertexCount, int& textureCount, int& normalCount, int& faceCount)
{
	ifstream fin;
	char input;


	// Initialize the counts.
	vertexCount = 0;
	textureCount = 0;
	normalCount = 0;
	faceCount = 0;

	// Open the file.
	fin.open(filename);

	// Check if it was successful in opening the file.
	if(fin.fail() == true)
	{
		return false;
	}

	// Read from the file and continue to read until the end of the file is reached.
	fin.get(input);
	while(!fin.eof())
	{
		// If the line starts with 'v' then count either the vertex, the texture coordinates, or the normal vector.
		if(input == 'v')
		{
			fin.get(input);
			if(input == ' ') { vertexCount++; }
			if(input == 't') { textureCount++; }
			if(input == 'n') { normalCount++; }
		}

		// If the line starts with 'f' then increment the face count.
		if(input == 'f')
		{
			fin.get(input);
			if(input == ' ') { faceCount++; }
		}

		// Otherwise read in the remainder of the line.
		while(input != '\n')
		{
			fin.get(input);
		}

		// Start reading the beginning of the next line.
		fin.get(input);
	}

	// Close the file.
	fin.close();

	return true;
}

bool LoadDataStructures(char* filename, int vertexCount, int textureCount, int normalCount, int faceCount)
{
	VertexType *vertices, *texcoords, *normals;
	FaceType *faces;
	ifstream fin;
	int vertexIndex, texcoordIndex, normalIndex, faceIndex, vIndex, tIndex, nIndex;
	char input, input2;
	ofstream fout;
```

```
// Initialize the four data structures.
vertices = new VertexType[vertexCount];
if(!vertices)
{
	return false;
}

texcoords = new VertexType[textureCount];
if(!texcoords)
{
	return false;
}

normals = new VertexType[normalCount];
if(!normals)
{
	return false;
}

faces = new FaceType[faceCount];
if(!faces)
{
	return false;
}

// Initialize the indexes.
vertexIndex = 0;
texcoordIndex = 0;
normalIndex = 0;
faceIndex = 0;

// Open the file.
fin.open(filename);

// Check if it was successful in opening the file.
if(fin.fail() == true)
{
	return false;
}

// Read in the vertices, texture coordinates, and normals into the data structures.
// Important: Also convert to left hand coordinate system since Maya uses right hand coordinate system.
fin.get(input);
while(!fin.eof())
{
	if(input == 'v')
	{
		fin.get(input);

		// Read in the vertices.
		if(input == ' ')
		{
			fin >> vertices[vertexIndex].x >> vertices[vertexIndex].y >>
			vertices[vertexIndex].z;

			// Invert the Z vertex to change to left hand system.
			vertices[vertexIndex].z = vertices[vertexIndex].z * -1.0f;
			vertexIndex++;
		}

		// Read in the texture uv coordinates.
		if(input == 't')
		{
			fin >> texcoords[texcoordIndex].x >> texcoords[texcoordIndex].y;
```

```
                        // Invert the V texture coordinates to left hand system.
                        texcoords[texcoordIndex].y = 1.0f - texcoords[texcoordIndex].y;
                        texcoordIndex++;
                }

                // Read in the normals.
                if(input == 'n')
                {
                        fin >> normals[normalIndex].x >> normals[normalIndex].y >>
                        normals[normalIndex].z;

                        // Invert the Z normal to change to left hand system.
                        normals[normalIndex].z = normals[normalIndex].z * -1.0f;
                        normalIndex++;
                }
        }

        // Read in the faces.
        if(input == 'f')
        {
                fin.get(input);
                if(input == ' ')
                {
                        // Read the face data in backwards to convert it to a left hand system from right hand system.
                        fin >> faces[faceIndex].vIndex3 >> input2 >> faces[faceIndex].tIndex3 >>
                        input2 >> faces[faceIndex].nIndex3 >> faces[faceIndex].vIndex2 >> input2 >>
                        faces[faceIndex].tIndex2 >> input2 >> faces[faceIndex].nIndex2 >>
                        faces[faceIndex].vIndex1 >> input2 >> faces[faceIndex].tIndex1 >> input2 >>
                        faces[faceIndex].nIndex1;
                        faceIndex++;
                }
        }

        // Read in the remainder of the line.
        while(input != '\n')
        {
                fin.get(input);
        }

        // Start reading the beginning of the next line.
        fin.get(input);
}

// Close the file.
fin.close();

// Open the output file.
fout.open("model.txt");

// Write out the file header that our model format uses.
fout << "Vertex Count: " << (faceCount * 3) << endl;
fout << endl;
fout << "Data:" << endl;
fout << endl;

// Now loop through all the faces and output the three vertices for each face.
for(int i=0; i<faceIndex; i++)
{
        vIndex = faces[i].vIndex1 - 1;
        tIndex = faces[i].tIndex1 - 1;
        nIndex = faces[i].nIndex1 - 1;

        fout << vertices[vIndex].x << ' ' << vertices[vIndex].y << ' ' << vertices[vIndex].z << ' '
             << texcoords[tIndex].x << ' ' << texcoords[tIndex].y << ' '
             << normals[nIndex].x << ' ' << normals[nIndex].y << ' ' << normals[nIndex].z << endl;

        vIndex = faces[i].vIndex2 - 1;
```

```
                tIndex = faces[i].tIndex2 - 1;
                nIndex = faces[i].nIndex2 - 1;

                fout << vertices[vIndex].x << ' ' << vertices[vIndex].y << ' ' << vertices[vIndex].z << ' '
                     << texcoords[tIndex].x << ' ' << texcoords[tIndex].y << ' '
                     << normals[nIndex].x << ' ' << normals[nIndex].y << ' ' << normals[nIndex].z << endl;

                vIndex = faces[i].vIndex3 - 1;
                tIndex = faces[i].tIndex3 - 1;
                nIndex = faces[i].nIndex3 - 1;

                fout << vertices[vIndex].x << ' ' << vertices[vIndex].y << ' ' << vertices[vIndex].z << ' '
                     << texcoords[tIndex].x << ' ' << texcoords[tIndex].y << ' '
                     << normals[nIndex].x << ' ' << normals[nIndex].y << ' ' << normals[nIndex].z << endl;
        }

        // Close the output file.
        fout.close();

        // Release the four data structures.
        if(vertices)
        {
                delete [] vertices;
                vertices = 0;
        }
        if(texcoords)
        {
                delete [] texcoords;
                texcoords = 0;
        }
        if(normals)
        {
                delete [] normals;
                normals = 0;
        }
        if(faces)
        {
                delete [] faces;
                faces = 0;
        }

        return true;
}
```

Summary



We can now convert .obj files into our simple model format.

Exercise

1. Add a .obj parser to your project that can load 3D models from .obj files without using the .obj converter.
2. Search or create three 3D models (something complicated, different from simple primitive models such as cube, cylinder, pyramid, etc…) and save them as .obj files. Render them on the same scene by loading from .obj files.
3. All models should be properly textured and shaped differently.

[Original script: www.rastertek.com]